A Distributed Stream Library for Java 8

Yu Chan, Andy Wellings, Ian Gray and Neil Audsley

Abstract—Java 8 has introduced new capabilities such as lambda expressions and streams which simplify data-parallel computing. However, as a base language for Big Data systems, it still lacks a number of important capabilities such as processing very large datasets and distributing the computation over multiple machines. This paper gives an overview of the Java 8 Streams API and proposes extensions to allow its use in Big Data systems. It also shows how the API can be used to implement a range of standard Big Data paradigms. Finally, it compares performance with that of Hadoop and Spark. Despite being a proof-of-concept implementation, results indicate that it is a lightweight and efficient framework, comparable in performance to Hadoop and Spark, and is up to 5 times faster for the largest input sizes tested.

Index Terms—Big Data, Java, distributed computing, programming models

1 Introduction

Recently, the 8th release of Java added support for aggregate data operations to the language. This paper identifies a number of weaknesses that are encountered when attempting to apply this to the programming of very large scale, and distributed, applications. It proposes a significant extension to the Java 8 streaming model to handle distributed data processing and storage.

Big Data is the term used for a collection of large and complex applications that are difficult to process using traditional data processing techniques, either due to the sheer scale of the data being produced, or tight timing requirements placed on the processing of the data. For example, the Large Hadron Collider can output a raw data stream of approximately 1PB/s [1]. This must be filtered before storage, placing a timing requirement on the filtering and storage stages. In some domains, requests for analytics and mining of stored datasets must be serviced sufficiently fast for the end user. In these situations, faster processing allows for potentially greater accuracy (i.e. by covering a larger historical dataset).

Java is the base language for a number of popular Big Data frameworks, including Hadoop [2], Spark [3] and Storm [4]. The introduction of streams and lambda expressions in Java 8 has brought Big Data closer to the core language. Java 8 streams allow programmers to view data processing in terms of pipelines of operations, and complementing it are lambda expressions that simplify functional programming. However, this stream-based programming model is still not sufficient for Big Data for two main reasons. Firstly, streams are limited to computations within a single machine, and there is no mechanism for distributing computations over multiple machines. This is due to streams being built on Java's Executor framework, which has no concept of distribution. Secondly, Java's default stream sources (such as collections and arrays) either store their data in-memory and thus cannot support very large datasets, or are not optimised for data-parallel computation.

In this paper, we address the above-mentioned issues and extend the programming model with Distributed Streams (which introduce the concept of distributed computing on a cluster) and Distributed and/or Stored Collections (which introduce the concepts of accessing very large datasets on-demand and of datasets that are distributed over a cluster).

Section 2 gives the background and motivation of Big Data computing in Java, and outlines the Java 8 Stream API. Section 3 details the requirements of Big Data computing and how Java 8 Streams fall short of these requirements. Our proposed extensions to the existing programming model and API are presented in Section 4, and our prototype implementation is described in section 5. Section 6 shows how MapReduce can be implemented in the proposed model while section 7 covers Spark and Storm. Section 8 evaluates these approaches, section 9 discusses related work and finally, section 10 concludes.

2 Background

The most popular open-source Big Data frameworks are targeted towards Java and/or higher-level languages that run on the JVM (eg. Scala). This is mainly because of the write-once-run-anywhere nature of JVM bytecode making it easier to deploy code on heterogeneous clusters. In this section, we review the popular Big Data frameworks Hadoop, Spark and Storm, in particular comparing their respective programming models.

The recent introduction of Java 8 has brought the language a step closer to having native Big Data capabilities due to the addition of the following:

- Java 8 Streams [5] provide a new programming model for data-parallel computation on a single machine. On multicore hardware it is able to parallelise a computation over multiple threads to speed up execution.
- Lambda expressions [6] simplify functional programming in Java, allowing concise instantiation of functional interfaces instead of using anonymous class syntax. These are used extensively in Java 8 Streams, making its code more readable.

Section 2.3 provides more details on Java 8 Streams.

2.1 Hadoop and MapReduce

Hadoop [2] is a prominent open source implementation of MapReduce [7], a popular batch processing model for Big Data computations. Such a model allows computations on a dataset that is typically partitioned over a cluster of nodes.

Before a MapReduce computation takes place, the required data needs to be distributed over the cluster. Hadoop achieves this with the *Hadoop Distributed File System* (HDFS) [8], which is overlaid on the host file system and manages the distribution of datasets across a cluster. Files in HDFS are divided into blocks (usually of 64MB) and replicated on different nodes by default. To avoid data coherency issues, files can only be written to once.

MapReduce requires programmers to implement *mappers* and *reducers*, each of which normally run on a node in the cluster. With reference to Hadoop, a MapReduce computation consists of three stages:

1) The *map* stage: Input data from a given file in HDFS is processed by a set of mappers, which output key-value pairs as a result.
2) The *shuffle* stage: The key-value pairs are collected over a set of reducers, with the same keys sent to the same reducer. For each key, the values are collected and sorted in a key-*value-list* pair.
3) The *reduce* stage: Each reducer processes all given key-*value-list* pairs and outputs the result in a local HDFS file. The full result of the computation is the concatenation of all partial results.

The number of mappers and reducers do not have to be equal. If the dataset is spread across the mapper nodes, Hadoop can optimise execution times by having each mapper node work on its local data.

2.2 Spark, Storm and stream-based processing

Though suitable for many applications, the MapReduce model is inflexible due to the fixed stage sequence. More recently, focus has shifted to in-memory stream-based models for Big Data processing, with the Spark [3] and Storm [4] frameworks being prominent examples of such models.

Spark removes some of MapReduce's limitations by exposing a pipeline-based programming model for data processing. Instead of implementing mappers and reducers, programmers specify a pipeline of operations on a *Resilient Distributed Dataset* (RDD). An operation either returns a new RDD (a *transformation*, which can be chained) or returns a value (an *action*, which terminates the pipeline).

Storm executes *topologies* which run indefinitely on a cluster. A topology defines a directed acyclic graph of nodes and data *streams* (vertices). A stream consists of an unbounded sequence of tuples. A node is either a *spout* (data source – a Twitter feed, for example) or a *bolt* (consumes and processes streams, and may emit new streams).

2.3 Java 8 Streams

A Java 8 *stream* is a sequence of data elements that can be processed by a *pipeline* of operations. Streams can be generated from several sources, including:

- Collections, by calling the `stream` method if the desired stream should be sequential, or the `parallelStream` method for a parallel stream;
- Arrays, by calling the `Arrays.stream` method;
- Factory methods in the `Stream` class;
- Files, by calling the `BufferedReader.lines` method.

After retrieving the stream from a source, a pipeline of aggregate operations can be performed on it, consisting of zero or more intermediate operations followed by a terminal operation. From the programmer's perspective, an intermediate operation returns a new stream of processed elements from the given stream, and a terminal operation returns a non-stream result.

Streams, pipelines and operations have the following properties and restrictions:

- Pipelines are evaluated lazily. The terminal operation triggers computation of the pipeline.
- Pipelines are short-circuiting. Only enough elements are consumed as required by the terminal operation.
- Pipelines are linear. There is a single stream source, and there is no branching mechanism for routing elements to different downstream operations.
- Streams can be traversed at most once. To use the data source again, a new stream has to be created.
- Operations should not change the data source if the source does not support concurrent modification.

Java 8 Streams further classify intermediate operations as stateless and stateful, depending on whether the operation needs to hold any state as data passes through. For example, `map` is a stateless operation as each element can be processed independently of another. However, the `distinct` operation (remove all duplicate elements from the Stream) is stateful because it must keep track of all encountered elements.

Details of the Java 8 Stream API can be found at [9].

2.4 Big Data example: word-count

An example of a Big Data computation is word-count, which will be used throughout the paper to illustrate related concepts. The simplest form of word-count is to count the number of words in a given input file. This can be expressed in Java 8 as shown in figure 1.

3 Big Data in Java 8

In this section we identify the necessary attributes of Big Data programming models, and then discuss the shortcomings of the Java 8 programming model with respect to these attributes and how these shortcomings can potentially be solved using a distributed stream model. We take an incremental approach to introducing Big Data in Java 8, as many projects start out small and realise the need for Big Data in later stages. For such cases, it may be easier to use a backward-compatible Big Data framework instead of rewriting programs using a very different programming model.

One of the most important attributes of Big Data programming models is the efficient processing of very large

```
import java.nio.file.*;
import java.util.*;
import java.util.regex.*;
import java.util.stream.*;
public class WordCount {
  public static long wordcount(Stream<String> lines) {
    Pattern delim = Pattern.compile("\\s+");
    return lines
      .flatMap(line -> Stream.of(delim.split(line)))
      .count();
  }
  public static void main(String[] args) {
    Path filename = Paths.get(args[0]);
    Stream<String> s = Files
      .lines(filename)
      .parallel();
    System.out.println(wordcount(s));
  }
}
```

Fig. 1. Simple word-count application using Java 8 Streams.

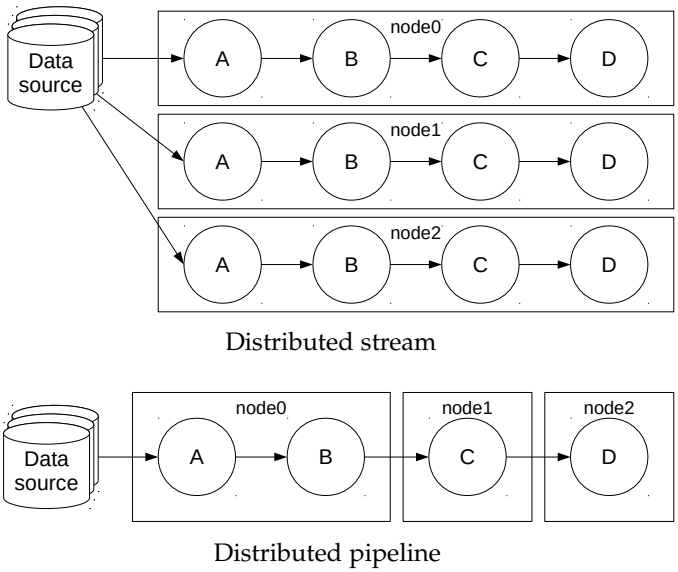

Fig. 2. Conceptual model of a distributed stream (top) and a distributed pipeline (bottom). Boxes indicate compute nodes, and the arrows indicate data flow from data source to pipeline operations.

datasets. This usually implies some form of data distribution mechanism as these datasets often do not fit in a single machine. For example, Hadoop uses its HDFS file system to transparently spread large files across the cluster. Spark, on the other hand, has wider support of data sources from text files on the local filesystem to distributed filesystems (including HDFS, Cassandra and Amazon S3) [10].

Data locality is also an important Big Data attribute. Since it is expensive to move data between nodes in a cluster, data needs to be kept local as much as possible. Nevertheless, Big Data programming models also provide a mechanism for moving data for cases where this is necessary (for aggregation, for example). Hadoop, which implements the MapReduce programming model, has a shuffle stage where all key-value pairs emitted from mappers are sent to the appropriate reducer depending on the key. In Spark, the movement of data (if any) is determined by the transformations and actions specified in the pipeline. In Storm, data locality is maintained by splitting each spout or bolt into tasks, running them over the nodes in a cluster and keeping the data streams between tasks local as much as possible.

3.1 Issues with the Java 8 programming model

There are three main problems with using Java 8 Streams as a Big Data programming model:

1) Java 8 Streams exist within a single JVM, and JVMs tend to only support individual SMP or ccNUMA machines, therefore stream computations will be limited to individual machines. There is no concept of a cluster and hence of distributing data or computations to other nodes in a cluster. Distribution requires the explicit use of middleware, such as MPI or an RPC framework. A difficulty of large-scale computation is that it requires support for both distribution of computation and distribution of data. Both methods potentially speed up processing, depending on the workload. An I/O-bound workload will see a speed improvement when distributing data, whilst a CPU-bound workload will benefit more from distributed computation.
2) Only in-memory datasets are well supported (by the Java collections framework), but these are not useful for very large datasets as it implies loading the entire dataset into memory. Datasets from existing on-disk sources (eg. `BufferedReader.lines`) are not optimised for large amounts of data and are likely to cause out-of-memory errors with large files. Furthermore, there is no concept of a dataset that is distributed over a cluster.
3) Using short-circuiting evaluation in a distributed computing model may have performance implications. Nodes will need to check if each data item they are working on is required by the terminal operation which may be on another node.

When considering a distributed programming model for Java 8 Streams, it should be noted that streams and pipelines can be distributed among compute nodes independently (see figure 2). A data source in a *distributed stream* is spread over or accessed by multiple nodes, while operations in a *distributed pipeline* span multiple nodes. Both methods potentially speed up processing, depending on the workload. An I/O-bound workload will see a speed improvement on a distributed stream, while a CPU-bound workload will benefit more on a distributed pipeline.

4 The Distributed Stream framework

In order to fulfill the distributed computing requirements of Big Data programming models in Java outlined in section 3, we introduce the Distributed Stream framework.

Central to this framework is the concept of *Distributed Streams*. A Distributed Stream facilitates parallelism by having a replicated pipeline that operates on different parts of a dataset, which is usually located in multiple nodes. There are several reasons for using Distributed Streams:

- The dataset is too large to fit on disk in one node.

```
package dstream;

public class ComputeNode {
  public String getName();
  public boolean isSelf();
  public static ComputeNode getSelf();
  public static ComputeNode findByName(String name);
}

public class ComputeGroup extends List<ComputeNode> {
  public ComputeGroup(Collection<ComputeNode> nodes);
  public ComputeGroup(ComputeNode node);
  public static ComputeGroup getCluster();
}
```

Fig. 3. The `ComputeNode` and `ComputeGroup` classes.

- It is too slow to read the entire dataset from a single node (due to limited I/O bandwith, for example).
- The dataset is located in storage nodes with specialised hardware, allowing fast parallel data access from a cluster.

The Distributed Stream framework does not limit an implementation to any particular programming paradigm. Thus, an implementation may run identical programs on every compute node which co-ordinate via the transport layer (the single-program-multiple-data or SPMD paradigm, which the prototype implementation uses), or it may utilise a driver node to distribute work to other nodes (the driver-worker or master-slave paradigm). The rest of this section describes the framework's components in greater detail.

4.1 Compute nodes and groups

Before distribution of data and computation can be supported, a method of identifying and grouping compute nodes is needed for decisions such as which node to send data to. We therefore define the `ComputeNode` class, which represents a compute node in the cluster, and the `ComputeGroup` class, which represents a group of compute nodes (see figure 3).

At program startup, all `ComputeNode` objects are initialised by the framework. Each compute node has a unique name (the model does not prescribe a node naming convention). There are also methods to get the current node or a specific node by its name.

The `ComputeGroup` class has constructors that accept individual compute nodes or a collection of them, as well as a static `getCluster` method to retrieve the entire cluster. Standard Java `List` methods can be called to modify the group's members.

If more functionality such as fault tolerance is needed, implementers can define new types of compute groups which extend `ComputeGroup`. These may monitor the status of nodes and amend the list of compute nodes as required.

The use of compute nodes and groups is optional as in many cases, data is partitioned across the entire cluster and computation therefore has to occur on all nodes.

```
package dstream.util;

public interface DistributedCollection<E>
    extends Collection<E> {
  public ComputeGroup getComputeGroup();
  public DistributedStream<E> stream();
  public DistributedStream<E> parallelStream();
  public static DistributedCollection<E> wrap(
    Collection<E> c, ComputeGroup grp);
  public static DistributedCollection<E> wrap(
    Collection<E> c);
}
```

Fig. 4. The Distributed Collection interface.

4.2 Distributed Collections

While Distributed Streams handle the data processing aspect of Big Data computing, they need to work on distributed datasets. For this, we introduce the concept of a *Distributed Collection* as a data source for a Distributed Stream. A Distributed Collection encapsulates data that is partitioned across a cluster.

Previously [11], we proposed the concept of Stored Collections, which extend Java (in-memory) collections to efficiently read large datasets from a local disk. Distributed forms of in-memory collections and Stored Collections are useful for caching intermediate results from Distributed Streams. Thus, recomputing data is avoided at the expense of memory or disk space usage. To support this, it must be possible to save data to these collections from a Distributed Stream. Since it is not possible to include all the data formats that Stored Collections will read from, the interface was designed to be easily extensible.

Since maps are also part of Java's collection framework, distributed maps will also be a necessary extension.

The Distributed Collection interface is shown in figure 4. This extends a Java collection with the following:

- The overridden `stream` and `parallelStream` methods now return Distributed Streams.
- The `getComputeGroup` method returns information about participating compute nodes.
- The `wrap` methods create a new Distributed Collection by grouping normal (non-distributed) collections across nodes together.

4.3 Drop-in replacement

To maintain compatibility, Distributed Streams are a drop-in replacement for Java 8 Streams. We propose a new `DistributedStream` interface that extends the existing `Stream` interface, overriding methods that return `Stream`s with those that return `DistributedStream`s. A partial definition of the Distributed Stream interface is in figure 5.

Primitive-type Stream interfaces will have corresponding Distributed Stream interfaces (`DistributedIntStream`, `DistributedLongStream` and `DistributedDoubleStream`).

Returning to the word-count example, these drop-in replacement extensions allow code describing the pipeline to be identical in both Streams and Distributed Streams (see figures 1 and 6 for a comparison). In this case, the

```
package dstream;

public interface DistributedStream<T>
   extends Stream<T> {
  public DistributedStream<T> distinct();
  public DistributedStream<T> filter(
    Predicate<? super T> predicate);
  public <R> DistributedStream<R> flatMap(
    Function<? super T,? extends
    Stream<? extends R>> mapper);
  ...
}
```

Fig. 5. The Distributed Stream interface.

```
import java.util.regex.*;
import java.util.stream.*;
import dstream.*;
import dstream.util.*;

public class WordCount {
  public static long wordcount(Stream<String> lines) {
    Pattern delim = Pattern.compile("\\s+");
    return lines
      .flatMap(line -> Stream.of(delim.split(line)))
      .count();
  }

  public static void main(String[] args) {
    String filename = args[0];
    Collection<String> c = new
      DistributedStringStoredCollection(filename);
    Stream<String> s = c.parallelStream();
    System.out.println(wordcount(s));
  }
}
```

Fig. 6. Simple word-count application using Distributed Streams.

`lines` argument in the `wordcount` method will be a `DistributedStream`. The pipelines are also replicated on each participating compute node.

However, some operations require communication between nodes to ensure correctness of results (the `count` operation in this example). This is solved by overriding these operations so that they behave as expected. Thus, the `count` operation in Distributed Streams additionally sums up the partial results of each participating node and sends the total count back to the nodes. The `flatMap` operation, on the other hand, does not require inter-node communication for correct results and its implementation can thus be left unchanged.

Finally, making these operations work across a compute group introduces two further issues:

- For operations such as `collect`, a large result requires significant network communication to replicate data elements on each node. This is often undesirable or unnecessary.
- For cases where the programmer actually intends to obtain local results on each node (for example, to obtain a partial count for further computation), this is currently not possible.

To address the issues, we provide new local variants of these operations that handle data within a node (see figure 7 for a few of these operations). Both variants behave identically on single-node clusters, but the local operations are more efficient over multiple nodes and can thus be used to construct more efficient pipelines.

```
package dstream;

public interface DistributedStream<T>
   extends Stream<T> {
  ...
  public <R, A> R localCollect(
    Collector<? super T, A, R> collector);
  public <R> R localCollect(Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator,
    BiConsumer<R, R> combiner);
  public DistributedStream<T> localDistinct();
  public DistributedStream<T> localForEach(
    Consumer<? super T> action);
  ...
}
```

Fig. 7. Local operations in the Distributed Stream interface.

```
package dstream;

@FunctionalInterface
public interface Partitioner<T> {
  public int partition(T data);
}
```

Fig. 8. Partitioner interface.

4.4 Distribution of data and computation

Distributed Streams take data locality into account and process local data as far as possible, to avoid moving data around the network unnecessarily. However, at certain points in the pipeline, it may be required for data to be distributed and gathered over the compute nodes to do partitioning and aggregation, such as in the MapReduce shuffle stage. Java 8 Streams support gathering of data with the general-purpose terminal operations `collect` and `reduce`, but there are currently no operations for distributing data according to a given algorithm (as there is no requirement to do this in the current Java 8 Streams). Therefore, we introduce a set of operations called `distribute` which facilitate the transfer of data using inter-node communication.

The Distributed Stream model does not require that all participating compute nodes execute exactly the same operations in the evaluation of a pipeline. For example, a pipeline can read data on a given set of input nodes, filter it on a different set of compute nodes, and then store it on a further different set of output nodes. The `distribute` operation can thus be viewed as moving the evaluation of part of the pipeline from one subset of the target cluster to another. It is also essential for splitting a stream into multiple streams.

In order to support distribution of data across the cluster, it is first necessary to provide some standard interfaces that allow the programmer to specify how the data should be partitioned. Figure 8 shows the interface that allows user-specified partitioning of data. Similar interfaces for primitive-type partitioners for the appropriate primitive-type Distributed Streams are also defined.

```
package dstream;

public interface DistributedStream<T>
   extends Stream<T> {
  public DistributedStream<T> distribute();
  public DistributedStream<T> distribute(
    Partitioner<? super T> p);
  public DistributedStream<T> distribute(
    ComputeGroup grp);
  public DistributedStream<T> distribute(
    ComputeGroup grp, Partitioner<? super T> p);
  public DistributedStream<T> distribute(
    ComputeNode node);
  ...
}
```

Fig. 9. Methods for distributing data between compute groups.

The `partition` method accepts a data element and returns an index representing a node in the compute group. For ease of use, it is not important for the programmer to know the size of the compute group, so if the index is out of range it will be wrapped around by the framework. This makes the partitioner suitable for range partitioning if the compute group size is known, as well as load balancing methods such as hash-based partitioning.

With a method to partition data, we define several variants of the `distribute` method that transfer data from one compute group to another, as shown in figure 9.

Parameters change the behaviour of `distribute` as follows:

- If used without parameters, `distribute` sends data to the same compute group according to a default hash-based partitioner.
- If a partitioner is given, it is used in place of the default hash-based partitioner.
- If a compute group/node is given, data is partitioned and sent to that group/node instead. The nodes in the specified compute group do not have to be part of the initial compute group.

A `distribute` operation returns a new Distributed Stream consisting of the same data elements which may have been moved across nodes.

Distributed Streams inherit pipelines from Java 8 Streams, which are linear in nature. While adequate in most cases, this hinders the concise implementation of programs which are better expressed with non-linear pipelines. These limitations can be addressed by allowing the splitting and joining of pipelines. Hence, two new operations are introduced (see also figure 10) to support these actions:

- The `split` operation splits the current pipeline into a specified number of pipelines. Each resulting pipeline receives the same stream of data elements from the current pipeline.
- The `join` operation merges a number of pipelines with the current pipeline. The resulting pipeline receives data elements from all pipelines.

To preserve data locality, these operations do not transfer data across nodes. However, a subsequent `distribute` operation may be used to move the data to the desired locations.

```
package dstream;

public interface DistributedStream<T>
  extends Stream<T> {
    public DistributedStream<T>[] split(
    int numStreams);
    public DistributedStream<T> join(
    DistributedStream<T>... streams);
  ...
}
```

Fig. 10. The `split` and `join` operations for creating non-linear pipelines.

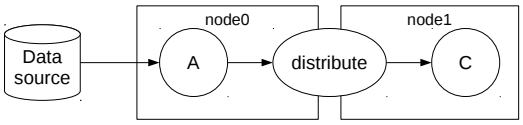

Fig. 11. A distributed pipeline to illustrate the problem of short-circuit evaluation in a distributed environment. Circles and ellipses represent pipeline operations, while arrows represent data flow.

A key property of Java 8 Streams is that they are short-circuiting, but extending this to a distributed pipeline environment impacts efficiency in the following ways:

- The pipeline has to wait for existing data items to be fully processed before deciding if more data items are needed. This may result in idle pipeline sections in certain nodes.
- Requests for data items originate from the terminal operation, and for distributed pipelines this has to be communicated upstream to different nodes which increases overhead. There is also the additional problem of choosing the node(s) from which to retrieve data.

Figure 11 shows an example pipeline that uses a `distribute` operation, and where operations *A* and (terminal) *C* are on different nodes. If the entire pipeline is short-circuiting, operation *A* will wait for a message from operation *C* to begin processing the next data item. The inefficiency is compounded if operation *A* potentially takes a long time to let a data item through the pipeline (eg. filtering). The time spent waiting in upstream operations could have been used to process data speculatively to keep those operations better utilised.

Consequentially, the solution employed is to view the `distribute` operation as terminating the current pipeline and starting another. This ensures that each section of the pipeline is on a single node and is evaluated as efficiently as possible without having to be concerned with parallelism in other pipeline sections. Therefore, this solution of restricting short-circuit evaluation to each node is simpler than enforcing it across nodes.

4.5 Summary

This section has detailed the four components in the Distributed Stream framework:

1) Compute nodes and compute groups for identifying and grouping nodes in a cluster.

2) Distributed Collections for representing distributed datasets and storing intermediate results.
3) Distributed Streams as a drop-in replacement for Java 8 Streams.
4) New eager operations for distributing data across nodes and support for splitting of streams.

5 Prototype implementation

This section gives an overview of the proof-of-concept framework we implemented. In general, it is based on the default Java 8 Stream implementation, inheriting and extending several of its classes. This allows for future changes to the default implementation without affecting Distributed Streams too significantly.

5.1 Transport layer

Since there is no requirement to use any particular transport layer, a patched version of the MPJ Express [12] library was used for several reasons:

- It is based on MPI, a message-passing protocol widely used in cluster-based parallel computing.
- It is lightweight and written in Java.
- It supports MPI in full multithreaded mode.

If required, the transport layer can be changed (e.g. to RMI) without significant modifications to the rest of the library.

Each compute node is identified by an integer identical to its global MPI communicator (`COMM_WORLD`) rank. The name of each node is the string "`node`" concatenated with its rank (in other implementations, this can be read from a configuration file, for example). The `ComputeNode` class in our implementation is also the program's entry point. The `main` method initialises MPI, discovers the other nodes in the cluster and runs the actual program. It also cleans up after the program finishes.

As defined in section 4.1, a `ComputeGroup` is a `List` of compute nodes. Thus it supports all `List` operations. The `getCluster` method creates a new compute group containing all nodes in the cluster as discovered by `ComputeNode.main`. The first node in each group is designated as the root.

The current prototype assumes no node failures. For a more robust implementation, this can be handled by configuring a timeout interval between messages sent to and received from a node, and retrying the computation, possibly on another node, if the timeout has been exceeded.

5.2 Distributed Collections

The `DistributedCollection` interface overrides the default methods `stream` and `parallelStream`. It also implements new static `wrap` methods which return a `WrappedDistributedCollection` of the same data type (see figure 12 for more details). It keeps a reference to the local collection, which is used when generating iterators and spliterators, or retrieving the local size of the collection.

```
package dstream.util;
...

public interface DistributedCollection<E>
  extends Collection<E> {
  public static <E> DistributedCollection<E> wrap(
    Collection<E> c, ComputeGroup grp) {
    return new WrappedDistributedCollection(c, grp);
  }

  public static <E> DistributedCollection<E> wrap(
    Collection<E> c) {
    return new WrappedDistributedCollection(c,
      ComputeGroup.getCluster());
  }

  ...
}

class WrappedDistributedCollection<E>
  extends AbstractCollection<E>
  implements DistributedCollection<E> {
  private ComputeGroup grp;
  private Collection<E> c;

  public WrappedDistributedCollection(
    Collection<E> c, ComputeGroup grp) {
    this.c = c;
    this.grp = grp;
  }

  @Override
  public ComputeGroup getComputeGroup() {
    return grp;
  }

  @Override
  public Iterator<E> iterator() {
    return c.iterator();
  }

  @Override
  public Spliterator<E> spliterator() {
    return c.spliterator();
  }
  ...
}
```

Fig. 12. Implementation of the `WrappedDistributedCollection` class and usage in `DistributedCollection.wrap` methods.

5.3 Distributed Streams – Distribution of data extensions

As mentioned in section 4.4, the `distribute` operation terminates the existing pipeline and starts a new one. The following algorithm describes the core of all `distribute` operation variants.

The existing pipeline is terminated with a `localForEach` operation, which sends each element to the appropriate destination node. (The MPI implementation serialises each element. Thus, our implementation can only send objects that implement the `Serializable` interface.) After all elements are sent, an end-of-data marker is sent to all participating nodes. We use a null object message to represent this marker.

Concurrently, a new pipeline is created and converts incoming messages into data elements. It determines that no more data is available when it has received end-of-data markers from all participating nodes.

Since stream computations block until the terminal operator has output a result, extra threads are created to keep the two pipelines executing concurrently.

5.4 Distributed Streams – Drop-in replacement extensions

We give details of how some of the Distributed Stream operations are implemented to satisfy the drop-in replacement requirement.

The `reduce` operation: There are a number of `reduce` operations for each stream type, but all have implementations similar to the following:

1) A reduction is performed on local data elements with the `localReduce` operation.
2) The local result is sent to the first node in the compute group.
3) The first node receives and accumulates all local results, and sends the final result to the other nodes.
4) The other nodes receive the final result. All nodes return the same value.

Unlike in MapReduce, where a reduce stage may output large amounts of data, a `reduce` operation combines data items into a single value. Again, with our simple prototype, we assume the nodes are reliable.

The `allMatch` operation can be expressed as a reduction with the result being the logical-AND of local `allMatch` operations on each participating node.

The `count` operation can be expressed as a reduction with the result being the sum of local `count` operations on each participating node.

The `distinct` operation can be expressed in terms of the `distribute` operation followed by the `localDistinct` operation. The `distribute` operation sends elements with the same hash value (which includes all identical elements) to the same node. The `localDistinct` operation then removes duplicate elements within each node.

The `forEach` operation: To be consistent with the other terminal operations, each participating node needs to perform the specified action on every element in the Distributed Stream. Thus, each node broadcasts its local data elements to other nodes, thereby ensuring that each local stream contains elements from all nodes. However, this is unlikely to be efficient. The `localForEach` operation avoids broadcasting elements and is intended for programmers to optimise their implementations.

6 MapReduce in Distributed Streams

Section 5 described how the API extensions are implemented. In this section we consider how the MapReduce programming model can be expressed in terms of the Distributed Stream model.

With Java 8 streams, performing MapReduce computations across a cluster was not possible as the framework operated within a single node. Distributed streams solve this problem by defining operations to transfer data across nodes. In MapReduce, the shuffle stage is where data transfer is needed. This stage can be broken down into sub-stages and implemented with distributed streams as follows:

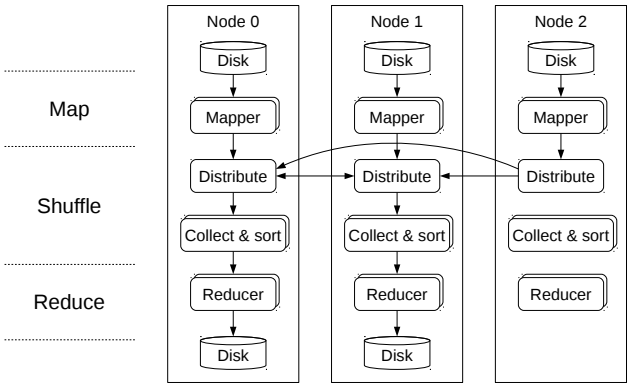

Fig. 13. Expressing a MapReduce computation in terms of distributed streams. Arrows indicate the flow of data. The mapper, reducer, local collect and local sort may span multiple stream operations.

1) The `distribute` operation transfers data (in key-value form) between compute nodes such that those with the same key are sent to the same node.
2) The `localCollect` operation on each node accumulates incoming data into a collection of key-*value-list* pairs. The value lists are optionally sorted. Since `localCollect` is a terminal operation, a new stream consisting of elements in the collection is created and passed to the reduce stage.

Figure 13 shows in general how a MapReduce computation can be represented with pipeline operations.

Special cases such as summing, counting and collecting are built into Distributed Streams. Hence the resulting code is more concise if such operations are used.

6.1 Illustrative example

We return to the word-count example introduced in section 2.4, however with a change in the problem specification to more effectively demonstrate how Distributed Streams can be used in MapReduce computations. This time a detailed word-count, which outputs a sorted list of words together with their frequencies, is used. Since Distributed Streams and Distributed Collections are drop-in replacements, the same algorithm can be used on an input text file distributed over the cluster, and the computation also occurs cluster-wide (see figure 14).

If only a subset of nodes are needed for reduction, the pipeline can be modified to distribute the elements to the required subset. Using the example above, the user need only make a small change to the pipeline by adding a `distribute` operation:

```
.distribute(lines.getComputeGroup().get(0))
```

between the `flatMap` and `collect` operations.

7 In-memory streaming in Distributed Streams

This section focuses on comparing the expressive power of Distributed Streams with Spark and Storm.

```
import java.util.regex.*;
import java.util.stream.*;
import dstream.*;
import dstream.util.*;

public class WordCountDetailed {
  public static void wordcount(
    DistributedCollection<String> lines) {
    Pattern pat = Pattern.compile("\\s+");
    Map<String, Long> words = lines
      // Generate a distributed, parallel stream
      .parallelStream()
      // Map stage
      .flatMap(line -> Stream.of(pat.split(line)))
      // Shuffle stage
      .collect(Collectors.groupingBy(
        w -> w, TreeMap::new,
        Collectors.counting()));
    Set<Map.Entry<String, Long>>
      entries = words.entrySet();
    entries
      // Generate a normal, sequential stream
      .stream()
      // Reduce stage
      .forEach(e -> {
        System.out.println(
          e.getKey() + "\t" + e.getValue()); });
  }

  public static void main(String[] args) {
    String filename = args[0];
    wordcount(new DistributedStringStoredCollection(
      filename));
  }
}
```

Fig. 14. Detailed word-count application using Distributed Streams.

7.1 Spark and comparisons with Distributed Streams

Since both Spark and Distributed Streams are stream-based programming models, there are some similarities. For example, Spark pipelines are also lazily evaluated, and their Java syntax resembles that of Java 8 streams. Also, transformations are analogous to intermediate operations and actions are equivalent to terminal operations.

To demonstate the similarities between Spark's transformations/actions and Distributed Streams' operations, table 1 lists a number of Spark transformations/actions together with the equivalent operations in Distributed Streams.

There are also significant differences between the models. Spark was designed specifically for Big Data applications, whereas Distributed Streams have to maintain compatibility with Java 8 Streams. RDDs can be reused, unlike Java 8 Streams, and Spark allows caching of data in memory for frequently-used RDDs to avoid recomputing data. Also, Java 8 Streams are conceptually separate from Java collections (which can be the source of a stream), but Spark RDDs do not have such a distinction. Depending on its position in the pipeline, an RDD may contain data from HDFS blocks, or transformed data cached in memory, or even information on how the data should be processed (to support lazy evaluation). This allows the entire pipeline to be lazy even across compute nodes.

Terminal operations in Distributed Streams use all-to-all communications by default, due to Java semantics. However, the group of nodes which receive the output of terminal operations can be changed from this default to any subset through the use of `distribute` and local versions of terminal operations. Spark's driver-worker paradigm limits all-to-all communications to its internal shuffle operations (which are often needed).

Spark, being a high-level framework, does not have operations for distributing data according to an arbitrary partitioner. However, it has operations such as `sample` and `intersection` that are less straightforward to implement in Distributed Streams. Spark also exposes the concept of data partitions to the programmer, while Distributed Streams do not. Thus, there are no equivalents for several of Spark's operations such as `coalesce` and `mapPartitions` which are exclusively for altering the number or location of partitions in the cluster.

7.2 Distributed Stream implementation for Spark

Due to the similarities in programming models, parts of the Spark and Distributed Stream pipelines can be almost identical. However, a Distributed Stream implementation will need to be aware of the following due to differences in the programming models:

- More Distributed Stream pipelines may be needed to implement a similar Spark pipeline. A terminal operation (except `distribute`) in Distributed Streams waits for all data in the stream to be processed before returning, and the next pipeline will not begin execution until this happens. This behaviour is inherited from Java 8 Streams. This may cause underutilisation of the nodes, and cannot be optimised by the implementation unlike Spark. To mitigate this, a separate thread can be created to execute another pipeline concurrently, but the programmer has to judge the feasibility of this (for example, to ensure that there are no data dependencies between the pipelines).
- A feature of Spark is that it can provide caching and reuse of computed values in a pipeline. This is not currently automated by Distributed Streams. The programmer must implement this manually through the use of `localCollect` and Distributed Collections.
- Since Spark has the concept of a driver program which defines the pipeline and submits work to the master node, actions such as `collect` send data to the driver instead of to all processes on participating nodes. To achieve a similar effect of sending all data to one compute node, the `distribute(`*`node`*`)` operation can be used.
- Like Distributed Stream operations, a number of Spark transformations and actions require significant inter-node communication when dealing with datasets spanning multiple nodes and should be avoided if there are still many elements in the dataset.
- If the underlying data source has no redundancy (we believe this is rare as common distributed file systems such as HDFS and Lustre [13] have this

TABLE 1
Comparison of Spark Transformations/Actions and Distributed Stream Operations.

Spark	Distributed Stream	Description
`.countByKey()`	`.collect(Collectors.groupingBy(e -> e.getKey(), Collectors.counting()))`	Counts the occurrence of each key and returns a map of key-count pairs.
`.filter(p)`	`.filter(p)`	Removes elements in the Distributed Stream that do not satisfy the predicate.
`.foreach(f)`	`.forEach(f)`	Execute a function over each element.
`.groupByKey()`	`.collect(Collectors. groupingBy(e -> e))`	Groups key-value pairs into key-*value-list* pairs.
`.map(f)`	`.map(f)`	Replaces each element with those from the mapping function.
`.reduce(f)`	`.reduce(f)`	Reduces the elements to a single value using a binary function.
`.take(n)`	`.limit(n).collect(Collectors.toList())`	Returns the first `n` elements in a `List`.

```
static void wordcount(JavaRDD<String> lines) {
  Pattern pat = Pattern.compile("\\s+");
  List<Tuple2<String,Integer>> result = lines
    .flatMap(line -> Arrays.asList(pat.split(line))
    // Convert to (word, 1)
    .mapToPair(s -> new Tuple2<String,Integer>(s, 1))
    // Add pairs with same words together
    .reduceByKey((i1, i2) -> i1 + i2)
    // Save as list of (word, count) pairs
    .collect();
  // Print each (word, count) pair
  for (Tuple2<?,?> t : result)
    System.out.println(t._1() + "\t" + t._2());
}
```

Fig. 15. Detailed word-count application using Spark.

property), the Distributed Collection will need to implement it.

7.3 Spark example

To demonstrate the similarities and differences between Spark and Distributed Streams, we refer to the detailed word-count example in section 6.1. The corresponding Spark code [14] (using Java 8) is shown in figure 15.

After replacing each line of text with the individual words (using `flatMap`), each word is converted to a *(word, 1)* pair. The `reduceByKey` transformation reduces values *(word, M)* and *(word, N)* to *(word, M + N)*. The `collect` action ends the pipeline, saving the remaining pairs into a `List`. Finally, the list contents are printed to standard output.

7.4 Storm and comparisons with Distributed Streams

Storm is also a stream-based programming model, but it is eager and emphasises task-parallel computations, setting it apart from Distributed Streams which are mainly lazy and data-parallel. Nodes in Storm normally run different computations which are then joined together with streams. Hence it is closer to a distributed pipeline model (see section 3) where each node processes a segment of the pipeline and does not have knowledge of the entire pipeline. On the other hand, a Distributed Stream is primarily replicated pipeline-based and makes use of compute groups to partition the cluster for different data-parallel computations.

```
public class Splitter implements IRichBolt {
  // Private variables are initialised in the
  // prepare() method
  private OutputCollector collector;
  @Override public vold execute(Tuple line) {
    Pattern pat = Pattern.compile("\\s+");
    // Send each word to next bolt
    for (String word: pat.split(line.getString(0)))
      collector.emit(new Values(word));
    collector.ack(line);
  }
  ...
}
public class Counter implements IRichBolt {
  // Private variables are initialised in the
  // prepare() method
  private OutputCollector collector;
  private TreeMap<String, Integer> wordcount;
  @Override public vold execute(Tuple t) {
    String word = t.getString(0);
    // Update occurrences or add new word
    if (wordcount.containsKey(word))
      wordcount.put(word, wordcount.get(word) + 1);
    else
      wordcount.put(word, 1);
    collector.ack(t);
  }
  ...
}
```

Fig. 16. Detailed word-count application using Storm.

7.5 Distributed Stream implementation for Storm

Since Storm uses a distributed pipeline model, the programmer can achieve a similar result by splitting the pipeline into several segments and executing the `distribute` operation at segment boundaries. For nodes that send data to multiple destinations, the `distribute` variant that accepts a collection of compute groups can be used.

7.6 Storm example

For Storm, we split the pipeline into two bolts shown in figure 16. The bolts can be incorporated into a topology with the code in figure 17.

When run, the `Splitter` accepts lines of text and outputs individual words. Concurrently, the `Counter` adds each word from the `Splitter` into a `TreeMap` or updates

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("input", ...);
builder.setBolt("split",
  new Splitter()).shuffleGrouping("input");
builder.setBolt("count",
  new Counter()).shuffleGrouping("split");
// ...
```

Fig. 17. Storm topology for detailed word-count application.

TABLE 2
Evaluated Workloads, Attributes and Input Sizes.

Workload	Attributes	Input sizes
Grep	Disk-intensive	2GB – 32GB
Sort	Disk- & comms-intensive	2GB – 32GB
Word-count (detailed)	Disk-intensive	2GB – 32GB
Bayes (naive)	Disk-intensive	1GB – 32GB
Connected components	Graph, comms-intensive	2^{10} – 2^{20} vertices
PageRank	Graph, comms-intensive	2^{10} – 2^{20} vertices

its occurrence if already present. Since Spark topologies run indefinitely, an external signal, special marker or timeout is needed to indicate the end of input and that the `TreeMap` is ready to be output.

8 Evaluation

The purpose of this evaluation is to demonstrate that the extended Distributed Stream APIs described in this paper allow efficient description of both map-reduce and distributed streaming processing.

Direct comparisons of end-to-end execution times for Distributed Streams, Spark and Hadoop are often not appropriate because each system performs different styles of computation, at different times, and with different data distribution, replication, and fault tolerance guarantees. Therefore this section attempts to remove initial distribution from the comparison and focus solely on the computation by pre-distributing the input data. This is how Hadoop (and Spark on HDFS) works normally, but it is not required by Storm or the Distributed Streams approach.

We based our evaluation on a subset of the BigDataBench benchmark suite [15]. Table 2 lists the workloads that were evaluated, along with their attributes and the input sizes used. For each workload, a Distributed Stream application was implemented and compared against the provided Hadoop and Spark implementations. Although the input sizes are smaller than standard Big Data workloads, we believe that they are sufficient for exploring the performance of the frameworks because the data is further broken down into smaller chunks for distribution and processing.

8.1 Experimental setup

All tests were carried out on a national, shared cluster[1] with the following allocation of nodes:

1) A master node and 10 compute nodes were used, with a total of 160 compute cores.
2) A master node and 20 compute nodes were used, with a total of 320 compute cores.

There were two Intel E5-2670 CPUs per node (for a total of 16 cores per node). Each node had 64 GB of RAM, an attached 7200RPM hard disk, and ran CentOS 6.7 Linux. All nodes were connected with InfiniBand. Hadoop version 1.2.1, Spark version 1.3.0 and Java 8u45 were used.

Using two cluster sizes allows evaluating the framework's scalability as more nodes are added. However, the shared nature of this cluster made measurements less accurate (there was the potential of interference from other users).

To minimise any differences in input data access, all tests read their input data from HDFS (as this was the default for the Hadoop and Spark tests) with data replication disabled. Input data was copied to HDFS before any measurements began, and the OS buffers were cleared before each test. The master node acted as the HDFS namenode as well as the jobtracker, while the compute nodes were each HDFS datanodes and tasktrackers.

In the BigDataBench suite, the Hadoop and Spark workloads read their data from HDFS. Thus, to use the same data source for all tests, a `HDFSStringCollection` was implemented for Distributed Stream tests. It reads data from HDFS as a `DistributedCollection` of lines when a stream is obtained. Since HDFS files are already partitioned across the cluster, each compute node reads from local file blocks to preserve data locality. For writing data, we implemented a `HDFSStringCollectionWriter` class that writes to a new HDFS file using the `add` method (since HDFS files can only be written to once).

8.2 Results and evaluation

The following metrics were obtained:

- Execution time – the elapsed time of the computation, excluding the time taken to copy data to HDFS.
- Network usage – the total volume of data transferred over the network during the computation, again excluding the initial copying of data to HDFS.

10 runs were carried out for each input size, and the average values and standard deviations were obtained. Figures 18 and 19 show the execution times and network usage for cluster 2 respectively. Distributed Streams scaled similarly well compared to Hadoop and Spark, with the results from cluster 1 following a similar pattern.

For disk-intensive workloads, we also analysed the disk usage of all frameworks and found that throughout the workload's execution, Distributed Streams had the highest disk read throughput and spent the least amount of time with disks idle. This is due to more optimised disk usage patterns. On each node, Distributed Streams allows only one task to read from disk at any time thereby keeping disk access mostly sequential, whilst the other frameworks do not. Thus, it was more efficient in reading data from disk and computation than the other frameworks. This behaviour cannot be changed in the other frameworks, as they leave these I/O optimisations to the operating system.

1. N8 High Performance Computing cluster (URL: http://n8hpc.org.uk)

Note that even though the tests were carried out as fairly as possible, Distributed Streams are a proof-of-concept and are not heavily optimised. On the other hand, Hadoop and Spark have been under active development for several years and are thus much more optimised. Fault tolerance is also not implemented in Distributed Streams. However we have removed fault tolerance as much as possible from the other systems and so we believe such overhead is not significant.

8.2.1 Execution time

The Distributed Stream implementations of disk-intensive workloads (Bayes, Grep, Sort and Word-count) ran significantly faster than or comparable to those for Hadoop and Spark.

For graph workloads (Connected components and PageRank), execution times for Distributed Stream implementations vary widely depending on the input size. Besides being less optimised, another reason is that Hadoop and Spark have optimised graph processing frameworks and libraries (Pegasus and GraphX) while the graph algorithms for Distributed Stream implementations were written from scratch. Despite these limitations, Distributed Streams were still fastest for the smallest datasets. It is likely that if more efficient graph algorithms and optimisations are used, they can scale as well as Hadoop and Spark for larger datasets.

8.2.2 Network usage

Hadoop and Spark attempt to load-balance tasks across the cluster, which may require nodes to read data on a non-local disk and have it sent across the network. Distributed Streams does not do this, except for sending data items that straddle HDFS blocks. This can be seen in results for the Grep and Word-count workloads, with network usage being higher and less predictable for Hadoop and Spark than Distributed Streams.

As previously mentioned, Distributed Streams are not heavily optimised. This is especially true of the communication layer. Though network usage of Distributed Streams was generally in the high end, they still ran as fast as (if not faster than) the other frameworks for non-graph workloads. This highlights the efficiency of Distributed Streams, and especially the Java 8 Stream framework that it builds upon.

8.3 Summary

Currently, Distributed Streams do not provide any fault tolerance above that which is already provided by the underlying communication layer implementation. Yet, these results illustrate that as a thin layer, the Distributed Streams API is a suitable extension to Java 8 Streams for Big Data computations.

With Distributed Streams being proof-of-concept, we believe there is room for efficiency improvements. To support larger clusters, reliability issues will also need to be addressed, and we are confident that performance will not be sacrificed as a result.

9 Related work

Big Data research and development has been dominated by the MapReduce and streaming programming models, which were discussed in section 2. This section gives a brief overview of other developments.

A Big Data framework which also uses Java 8 Streams has been proposed in [16]. It differs from our approach in that their extended stream model is used within other Big Data engines (such as Hadoop), while our framework proposes a new engine. Our approach does not require an existing engine for processing data, and is backward compatible with the existing model.

Other streaming models include the Cascading [17] framework, which implements a Java-based programming model with concepts borrowed from Unix pipes. Its main abstraction is the *flow*, which consists of pipelines of computation bound to data sources and sinks. This differs from Java 8 Streams in that the pipelines can be defined independent of data sources, while pipelines in Java 8 Streams must be constructed from a known data source.

There have also been attempts to improve existing programming models. For example, to address the inflexibility of the MapReduce model, frameworks such as Twister [18] have extended the model by adding iteration and caching of data across MapReduce jobs. This reduces the amount of data read during each iteration and thus speeds up execution. HaLoop [19] also modifies Hadoop to make it easier to build iterative applications, something that is difficult in the basic MapReduce model. HaLoop again aggressively caches data and results to reduce recalculation. CoHadoop [20] suggests a lightweight extension to the Hadoop programming model to allow control over the placement of data.

A combination of the above-mentioned approaches can be found in the Stratosphere [21] platform, which includes a high-level query language, Meteor, implemented on top of a generalised MapReduce programming model, PACT [22]. PACT extends MapReduce to support more user-defined functions (besides `map` and `reduce`), acyclic data flows and arbitrary length records (as opposed to key-value pairs in MapReduce).

Hazelcast [23] implements distributed versions of Java collections. However, they are in-memory only and require populating the collections before computation can begin.

10 Conclusions

In this paper we have proposed and implemented a proof-of-concept version of Distributed Streams. Distributed Streams provide the following features which address the limitations of the existing Java 8 Stream model:

- It provides compute nodes and groups to support computation across a cluster.
- It provides replicated and distributed pipelines for distributed stream-based processing, as well as new operations for the distribution of data across a cluster.
- It provides Distributed Collections to encapsulate data that is partitioned across a cluster, allowing them to be used as data sources for Distributed Streams.
- It is a drop-in replacement for Java 8 Streams, allowing programmers to easily port existing applications to run over a cluster.

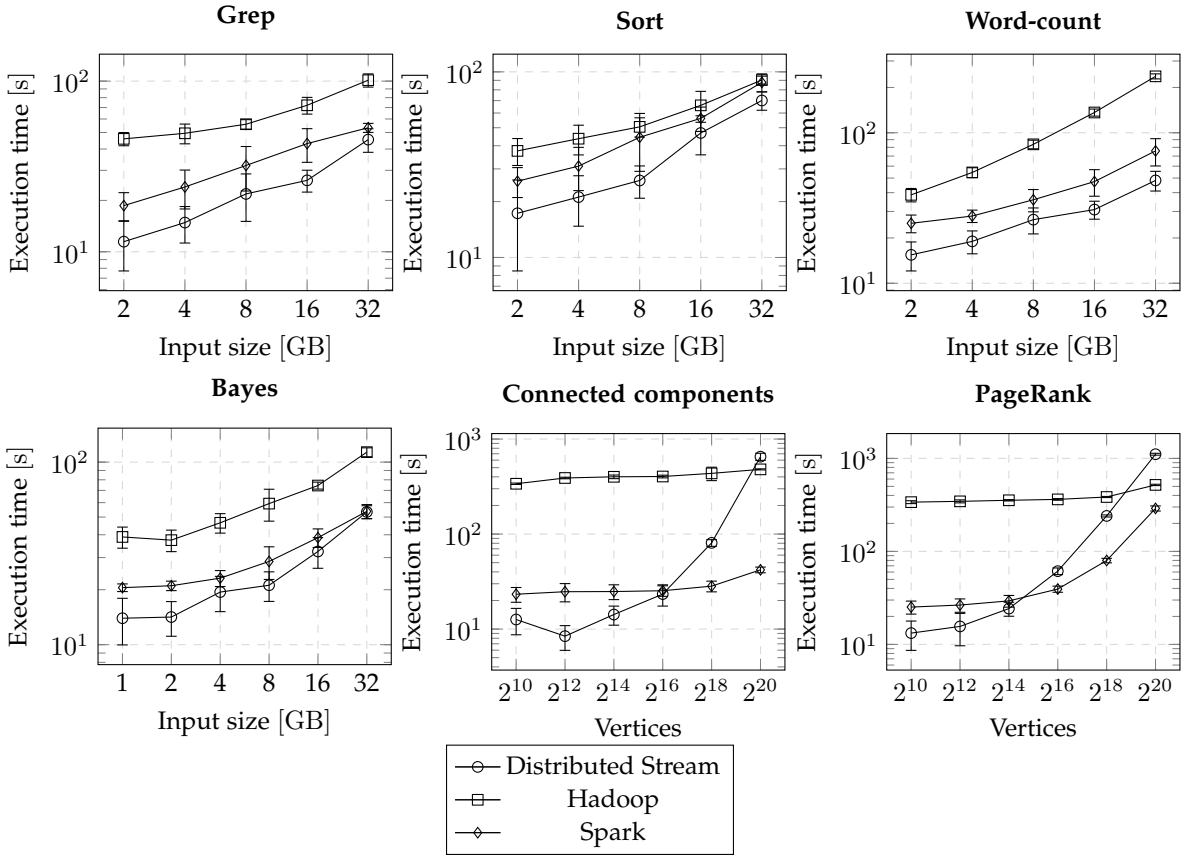

Fig. 18. Average execution time for each workload on cluster 2 (20 nodes).

Most of the Java 8 programming model was directly applicable to distribution. However the main difficulty observed was that short-circuit evaluation is very inefficient when distributed. Accordingly, a new operation had to be added to the Java model to allow for this. Otherwise, the proposed framework is a drop-in replacement for existing Java 8 code. In addition, pipelines have been made more flexible in Distributed Streams, allowing for more expressiveness in the programming model.

More generally, this work looked at taking a parallel programming model (Java 8 Streams) and making it distributed. This provides backward compatibility and allows programmers to incrementally optimise sections of their code. This is in contrast to most Big Data programming models which start out as distributed models.

Our results have shown that Distributed Streams can in many cases achieve similar execution times compared to Hadoop and Spark. This indicates that Java 8 Streams may be more effective at fully utilising the resources of compute nodes of the distributed system. In the future, we plan to further optimise the performance of Distributed Streams by integrating existing research on optimising network traffic and to address reliability issues by implementing SPMD fault tolerance in the communication layer and supporting dynamically changing clusters.

ACKNOWLEDGMENTS

This research is supported by the JUNIPER project, which has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration – grant number 318763. It also forms part of a PhD thesis [24]. We would like to thank the referees for their comments on earlier versions of the paper.

REFERENCES

[1] X. C. Vidal and R. C. Manzano, "Taking a closer look at LHC," http://www.lhc-closer.es/1/3/12/0, accessed 2014/05/05. [Online]. Available: http://www.lhc-closer.es/1/3/12/0

[2] Apache Software Foundation, "Apache Hadoop," http://hadoop.apache.org/, accessed 2013/09/01.

[3] ——, "Apache Spark – Lightning-Fast Cluster Computing," http://spark.incubator.apache.org/, accessed 2013/10/03.

[4] N. Marz, "Storm – Distributed and fault-tolerant realtime computation," http://storm-project.net/, accessed 2013/10/03.

[5] Oracle Corporation, "JEP 107: Bulk Data Operations for Collections," http://openjdk.java.net/jeps/107, accessed 2013/09/05. [Online]. Available: http://openjdk.java.net/jeps/107

[6] ——, "JSR 335: Lambda Expressions for the Java(TM) Programming Language," https://jcp.org/en/jsr/detail?id=335, accessed 2015/06/01. [Online]. Available: https://jcp.org/en/jsr/detail?id=335

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[8] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, 2009. [Online]. Available: http://books.google.co.uk/books?id=bKPEwR-Pt6EC

[9] Oracle Corporation, "Stream (Java Platform SE 8)," https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, accessed 2014/11/17.

[10] Apache Software Foundation, "Spark Programming Guide," http://spark.apache.org/docs/latest/programming-guide.html, accessed 2015/08/21.

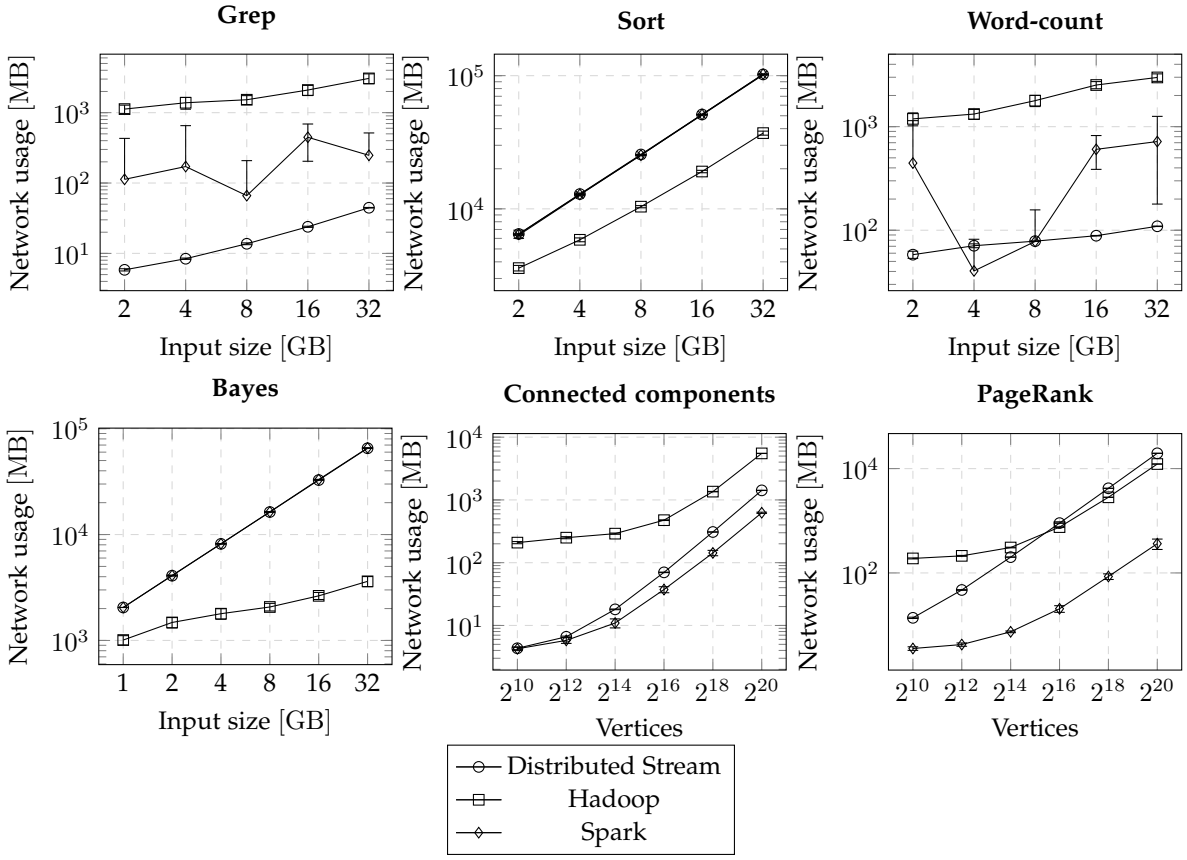

Fig. 19. Average network usage for each workload on cluster 2 (20 nodes).

[11] Y. Chan, I. Gray, and A. Wellings, "Exploiting Multicore Architectures in Big Data Applications: The JUNIPER Approach," in *Proceedings of 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, January 2014.

[12] B. Carpenter, A. Shafi, and M. Baker, "MPJ Express Project," http://www.mpjexpress.org/, accessed 2014/11/16.

[13] OpenSFS and EOFS, "Lustre," http://lustre.org/, accessed 2015/08/24.

[14] Apache Software Foundation, "Spark Examples," http://spark.apache.org/examples.html, accessed 2014/10/22.

[15] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 488–499.

[16] X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz, "Changing engines in midstream: A java stream computational model for big data processing," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1343–1354, Aug. 2014. [Online]. Available: http://dx.doi.org/10.14778/2733004.2733007

[17] Concurrent Inc., "Cascading — Application Platform for Enterprise Big Data," http://www.cascading.org/, accessed 2015/09/21.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[20] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: Flexible data placement and its exploitation in hadoop," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 575–585, Jun. 2011. [Online]. Available: http://dx.doi.org/10.14778/2002938.2002943

[21] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-014-0357-y

[22] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: A programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 119–130. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807148

[23] Hazelcast, Inc., "Hazelcast.org - The Leading Open Source In-Memory Data Grid," http://hazelcast.org/, accessed 2016/05/30.

[24] Y. Chan, "A Distributed Stream Library for Java 8," Ph.D. dissertation, University of York, 2016.

Yu Chan is a PhD student at the University of York, UK under the supervision of Ian Gray and Andy Wellings.

Andy Wellings is Professor of Real-Time Systems at the University of York, UK in the Computer Science Department.

Ian Gray is a Research Fellow at the University of York, UK in the Computer Science Department.

Neil Audsley is a Professor at the University of York, UK in the Computer Science Department.